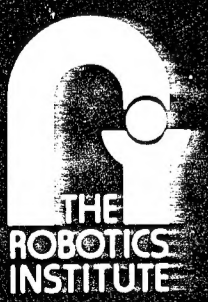
The D* Algorithm for Real-Time Planning of Optimal Traverses

Anthony Stentz

CMU-RI-TR-94-37



Carnegie Mellon University

The Robotics Institute

Technical Report

The D* Algorithm for Real-Time Planning of Optimal Traverses

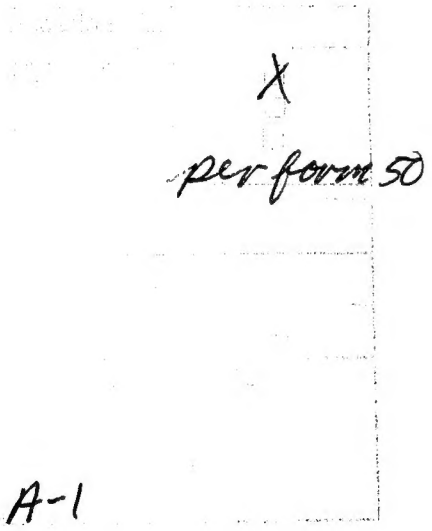
Anthony Stentz

CMU-RI-TR-94-37

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
September 1994

This research was sponsored by ARPA, under contracts "Perception for Outdoor Navigation" (contract number DACA76-89-C-0014, monitored by the US Army TEC) and "Unmanned Ground Vehicle System" (contract number DAAE07-90-C-R059, monitored by TACOM). Views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of ARPA or the United States Government.

Table of Contents



List of Figures

Abstract

Finding the lowest-cost path through a graph of states is central to many problems, including route planning for a mobile robot. If arc costs change during the traverse, then the remainder of the path may need to be replanned. Such is the case for a sensor-equipped mobile robot with incomplete or inaccurate information about its environment. As the robot acquires additional information via its sensors, it has the opportunity to revise its plan to reduce the total cost of the traverse. If the prior information is grossly incomplete or inaccurate, the robot may discover useful information in nearly every piece of sensor data. During the replanning process, the robot must either stop and wait for the new path to be computed or continue to move in the wrong direction; therefore, rapid replanning is essential. This paper describes a new algorithm, D*, capable of planning optimal traverses in real-time through focussed state expansion. D* repairs plans quickly by taking advantage of the fact that most arc cost corrections occur in the vicinity of the robot and the path needs only to be replanned out to the robot's current state. D* can be used not only for route planning but for any graph-based cost optimization problem for which arc costs change during the traverse of the solution path.

1.0 Introduction

The problem of path planning can be stated as finding a sequence of state transitions through a graph from some initial state to a goal state, or determining that no such sequence exists. The path is optimal if the sum of the transition costs, also called arc costs, is minimal across all possible sequences through the graph. If during the "traverse" of the path, one or more arc costs in the graph are discovered to be incorrect, the remaining portion of the path may need to be replanned to preserve optimality. A traverse is optimal if every transition in the traverse is part of an optimal path to the goal, assuming at the time of each transition, all known information about the arc costs is correct.

An important application for this problem, and the one that will serve as the central example throughout the paper, is the task of path planning for a mobile robot equipped with a sensor, operating in a changing, unknown or partially-known environment. The states in the graph are robot locations, and the arc values are the costs of moving between locations, based on some metric such as distance, time, energy expended, risk, etc. The robot begins with an initial estimate of arc costs comprising its "map", but since the environment is only partially-known or changing, some of the arc costs are likely to be incorrect. As the robot acquires sensor data, it can update its map and replan the optimal path from its current state to the goal. It is important that the replanning is fast, since during this time the robot must either stop or continue to move along a suboptimal path.

The field of path planning has received considerable attention in the research literature (see Latombe [5] for a good survey), but only recently has the problem of planning in unknown, partially-known, or dynamic environments been addressed. The algorithms can be divided into two classes. The first class consists of algorithms which are computationally fast and memory efficient, but which yield suboptimal traverses. Goto and Stentz [2] generate a "global" path using the known information and then attempt to "locally" circumvent obstacles on the route detected by the sensors. If the route is completely obstructed, a new global path can be produced. Lumelsky [7] initially assumes the environment to be devoid of obstacles and then moves the robot directly toward the goal. If an obstacle obstructs the path, the robot moves around the perimeter until the point on the obstacle nearest the goal is found. The robot then proceeds to move directly toward the goal again. Pirzadeh [9] adopts a strategy whereby the robot wanders about the environment until it discovers the goal. The robot repeatedly moves to the adjacent location with lowest cost and increments the cost of a location each time it visits it to penalize later traverses of the same space. Korf [4] uses initial map information to estimate the cost to the goal for each state and efficiently updates it with backtracking costs as the robot moves through the environment. These algorithms are computationally fast because the "replanning" operations are mostly local decisions about which way to move next. Little if any state information is retained during the traverse, so the algorithms are memory efficient. Because of the local scope of this replanning, however, completeness can be guaranteed but optimality cannot. This suboptimality can be extreme in some cases.

The second class of algorithms guarantees an optimal traverse but at greater computational and memory cost. One algorithm plans an initial path with A* [8] or the distance transform [3] using the prior map information, moves the robot along the path until either it reaches the goal or its sensors discover a discrepancy between the map and the environment, updates the map, and then replans a new path from the robot's current state to the goal. Although this brute-force replanner is optimal, it can be grossly inefficient, particularly in expansive environments where the goal is far away and little map information exists. Zelinsky [15] increases efficiency by using a quad-tree [11] to represent the free and obstacle space, thus reducing the number of states to search in the planning space. This approach is suboptimal or inefficient, however, if the state-to-state costs for moving through the environment range over a continuum.

Boult [1] maintains an optimal cost map from the goal to all states in the environment, assuming the environment is bounded (finite). When discrepancies are discovered between the map and the environment, the algorithm updates only the affected portion of the cost map. The map representation is limited to polygonal obstacles and free space. Trovato [14] and Ramalingam and Reps [10] extend this approach to handle graphs with arc costs ranging over a continuum. The limitation of this class of algorithms is that the entire affected portion of the map must be repaired before the robot can resume moving and subsequently make additional corrections to the map. Thus, the algorithms are inefficient when the robot is near the goal and the affected portions of the map have long "shadows".

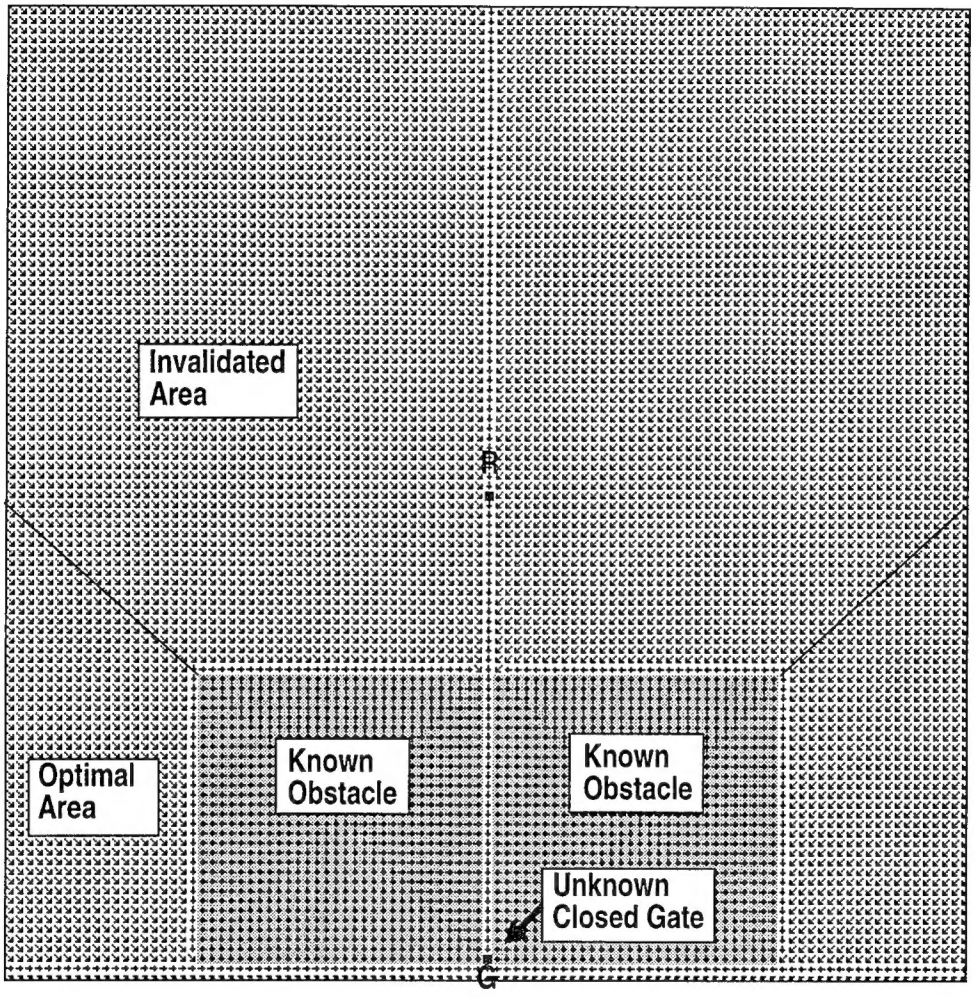
This paper presents a new algorithm, D*, for generating optimal traverses in real-time through a graph with changing or updated arc costs. D* takes advantage of the fact that most arc cost corrections occur in the vicinity of the robot, and only the portion of the path from these corrections out to the robot's current location needs to be replanned. D* maintains a partial, optimal cost map limited to those locations likely to be of use to the robot. Likewise, repair of the cost map is generally partial, re-entrant, and limited only to those states likely to yield a new, optimal path to the robot. D* establishes conditions for determining when the repair can be halted, either because a new path is found or the old one is still optimal. Thus, D* is very computationally and memory efficient, and it works in unbounded environments as well.

The algorithm is formulated in terms of a repeated, optimal find-path problem within a directed graph, where the arcs are labelled with transition cost values that can range over a continuum. Arc cost corrections (e.g., from a sensor) can be made at any time, and the known, measured, and estimated arc values comprise the map of the environment. The algorithm can be used for any planning representation, including visibility graphs [6] and grid cell structures. The paper begins with the basic form of the algorithm introduced by Stentz [13] and then describes new extensions to it that improve its computational and memory efficiency. Examples follow that illustrate the algorithm in operation. Experimental results are presented that compare several variations of D* to the brute-force replanner. Finally, conclusions are drawn.

2.0 The Basic D* Algorithm

The algorithm is named D* because it resembles A* [8], except that it is *dynamic* in the sense that arc costs can change during the traverse of the solution path. Provided that the traverse is properly coupled to the replanning process, it is guaranteed to be optimal. This section begins with the intuition behind the algorithm, defines the notation used, and presents D* as it was described in Stentz [13].

Figure 1: Invalidated States in the Graph

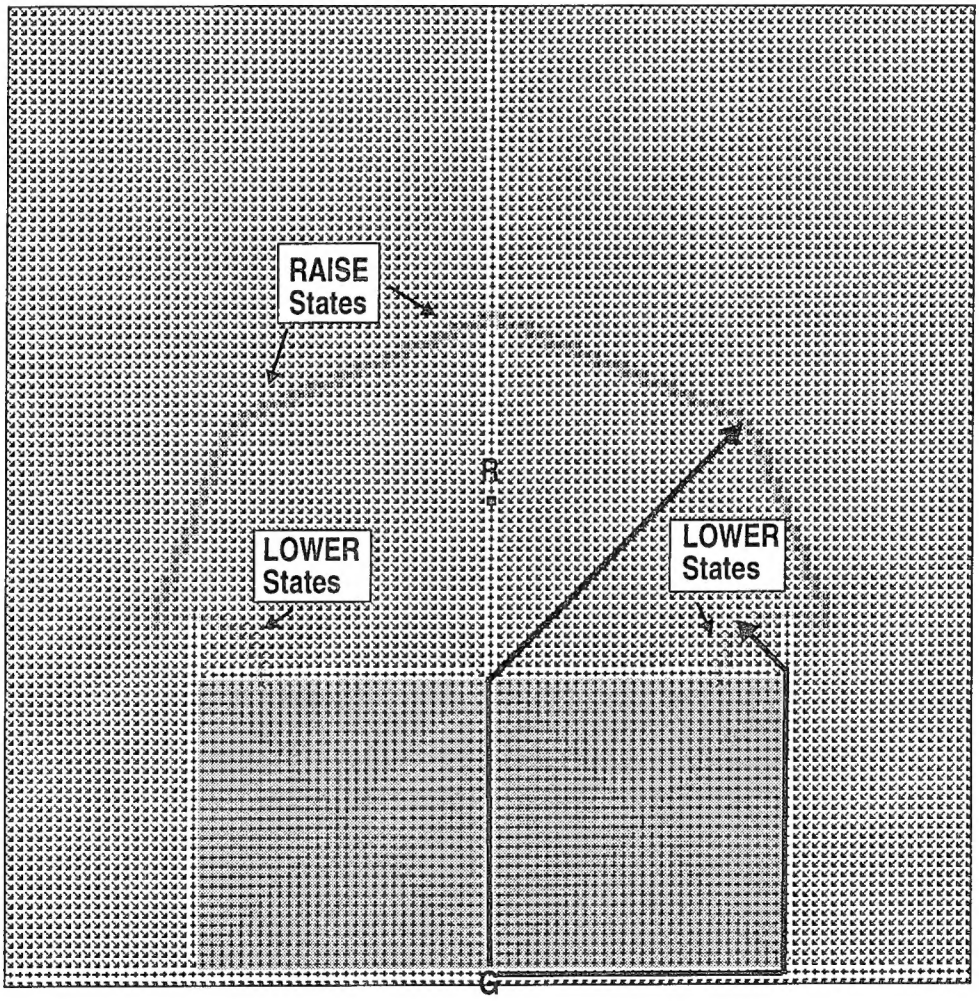


2.1 Intuition

In order to understand D*, consider how A* solves the following robot path planning problem. Figure 1 shows an eight-connected graph representing a Cartesian space of robot locations. The states in the graph, depicted by arrows, are robot locations, and the arcs encode the cost of moving between states. The white regions are locations known to be in free space. The arc cost for moving between free states is a small value denoted by $EMPTY$. The grey regions are known obstacle locations, and arcs connected to these states are assigned a prohibitively high value of $OBSTACLE$. The small black square is a closed gate believed to be open (i.e., $EMPTY$ value). The robot is point-sized and occupies only one location at a time. It should be noted that planning with non-zero size and shape can be reduced to a point planning problem [6]. Unfocussed A* can be used to compute optimal path costs from the goal, G, to all states in the space given the initial set of arc costs, as shown in the figure. The arrows indicate the optimal state transitions; therefore, the optimal path for any state can be recovered by following the arrows to the goal. Because the closed gate is assumed to be open, A* plans a path through it.

The robot starts at some initial location and begins following the optimal path to the goal. At location R, the robot's sensor discovers the gate between the two large obstacles is closed. This corresponds to an incorrect arc value in the graph: rather than *EMPTY*, it has a much higher value of *GATE*, representing the cost of first opening the gate and then moving through it. All paths through this arc are (possibly) no longer optimal, as indicated by the labelled region. A* could be used to recompute the cost map, but this is inefficient if the environment is large and/or the goal is far away. Another approach is to mark the affected states as invalid, place the neighboring, valid states on the *OPEN* list, and grow new optimal paths into the invalidated area through state expansion [1][10][14]. This approach is generally better than recomputing the cost map, but we are only interested in repairing the optimal paths out to the robot's current location, and then continuing to move the robot. The rest of the invalidated states need only be processed if the robot is forced to venture into that area; therefore, the repair should be incremental and re-entrant.

Figure 2: Propagation of *RAISE* and *LOWER* states

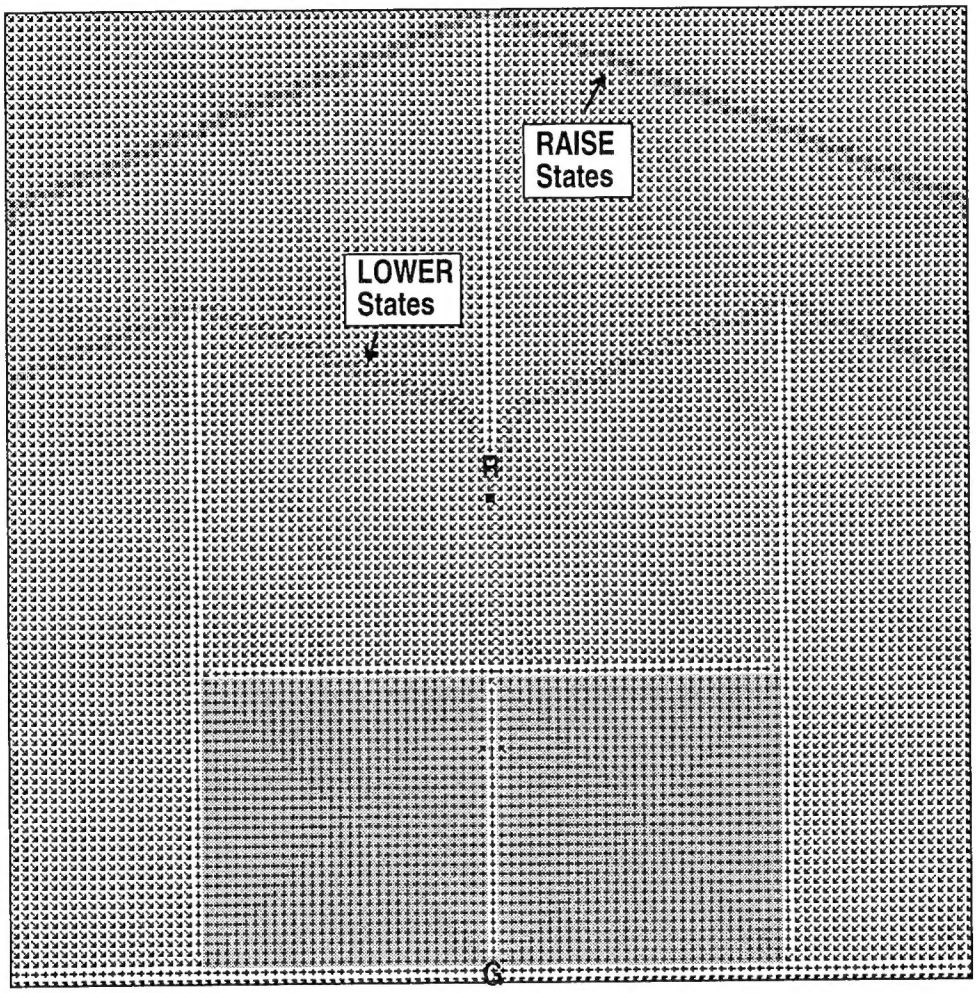


D* provides both features. Like A*, it maintains an *OPEN* list of states for expansion; however, these states consist of two types: *RAISE* and *LOWER* (see Figure 2). *RAISE* states transmit path cost increases due to an increased arc value, and *LOWER* states reduce costs and re-direct arrows to compute new optimal paths. The *RAISE* states propagate the arc cost increase through the invalidated states, adding the value of *GATE* to all paths passing through it. The *RAISE* states activate neighboring *LOWER* states which sweep in behind to reduce costs and re-direct pointers. *LOWER* states compute new, optimal paths to the states that were previously raised.

LOWER states are placed on the *OPEN* list by current path cost value (i.e., cost from the state to the goal), and the *RAISE* states are placed on the list by previous, unraised path cost value. States on the list are processed in the order

of increasing value. The intuition is that the previous optimal path costs of the *RAISE* states define a lower bound on the path costs of *LOWER* states they can discover. Thus, if the path costs of the *LOWER* states currently on the *OPEN* list exceed the previous path costs of the *RAISE* states, then it is worthwhile processing *RAISE* states to discover (possibly) a better *LOWER* state. Note in the figure that the arrow from the goal to the *RAISE* state wave front is equal in length to the arrow to the *LOWER* state wave front.

Figure 3: Repair of Graph out to Robot is Complete



The process can terminate when the lowest value on the *OPEN* list equals or exceeds the robot's path cost, since additional expansions cannot possibly find a better path to the goal. In Figure 3, the *LOWER* state propagation has reached the robot's location and a new optimal path has been computed to the goal. If the value of *GATE* had been smaller, the optimal path from the robot to the goal could have passed through the gate, rather than around the obstacles. In this case, the lowest value on the *OPEN* list would have equalled or exceeded the robot's raised path cost before the *LOWER* states reached the robot. Thus, the process would have terminated with the original path to the goal intact.

Once a new optimal path is computed or the old one is determined to be valid, the robot can continue to move toward the goal. Note in the figure that only part of the cost map has been repaired. It is possible that at a later point in the traverse, the robot will discover obstacles that force it back into the unrepaired, invalidated region. The *RAISE* and *LOWER* states placed on the *OPEN* list from these new obstacles will "mix" with the *OPEN* states from the previous, but not fully propagated, arc cost updates.

D* supports multiple, partially-propagated cost updates by sorting the states on the *OPEN* list by the *minimum* of all path cost values assumed by the state immediately before insertion on the *OPEN* list and while resident on the list. Thus, *RAISE* and *LOWER* states propagate cost increases and reductions, respectively, from the lowest to highest path cost value in the map, regardless of the order in which arc cost corrections are made and the extent to which they are propagated. It is even possible for *LOWER* states to become *RAISE* states and vice versa in the process. If the robot ventures into a previously invalidated region, the algorithm propagates all cost updates logged to that point into this region to compute a new optimal path to the goal.

2.2 Definitions

To formalize this intuition, we introduce the following notation and definitions. The problem space can be formulated as a set of *states* denoting robot locations connected by *directional arcs*, each of which has an associated cost. The robot starts at a particular state and moves across arcs (incurring the cost of traversal) to other states until it reaches the *goal* state, denoted by G. Every state X except G has a *backpointer* to a next state Y denoted by $b(X) = Y$. D* uses backpointers to represent paths to the goal. The cost of traversing an arc from state Y to state X is a positive number given by the *arc cost* function $c(X, Y)$. If Y does not have an arc to X, then $c(X, Y)$ is undefined. Two states X and Y are *neighbors* in the space if $c(X, Y)$ or $c(Y, X)$ is defined.

Like A*, D* maintains an *OPEN* list of states. The *OPEN* list is used to propagate information about changes to the arc cost function and to calculate path costs to states in the space. Every state X has an associated *tag* $t(X)$, such that $t(X) = NEW$ if X has never been on the *OPEN* list, $t(X) = OPEN$ if X is currently on the *OPEN* list, and $t(X) = CLOSED$ if X is no longer on the *OPEN* list. For each state X, D* maintains an estimate of the sum of the arc costs from X to G given by the *path cost* function $h(G, X)$. Given the proper conditions, this estimate is equivalent to the optimal (minimal) cost from state X to G. For each state X on the *OPEN* list (i.e., $t(X) = OPEN$), the *key* function, $k(G, X)$, is defined to be equal to the minimum of $h(G, X)$ before modification and all values assumed by $h(G, X)$ since X was placed on the *OPEN* list. The key function classifies a state X on the *OPEN* list into one of two types: a *RAISE* state if $k(G, X) < h(G, X)$, and a *LOWER* state if $k(G, X) = h(G, X)$. D* uses *RAISE* states on the *OPEN* list to propagate information about path cost increases and *LOWER* states to propagate information about path cost reductions. The propagation takes place through the repeated removal of states from the *OPEN* list. Each time a state is removed from the list, it is *expanded* to pass cost changes to its neighbors. These neighbors are in turn placed on the *OPEN* list to continue the process.

States on the *OPEN* list are sorted by their key function value. The parameter k_{min} is defined to be $min(k(X))$ for all X such that $t(X) = OPEN$. The parameter k_{min} represents an important threshold in D*: path costs less than or equal to k_{min} are optimal, and those greater than k_{min} may not be optimal. The parameter k_{old} is defined to be equal to k_{min} prior to most recent removal of a state from the *OPEN* list. If no states have been removed, k_{old} is undefined.

An ordering of states denoted by $\{X_1, X_N\}$ is defined to be a *sequence* if $b(X_{i+1}) = X_i$ for all i such that $1 \le i < N$ and $X_i \neq X_j$ for all (i,j) such that $1 \le i < j \le N$. Thus, a sequence defines a path of backpointers from X_N to X_1. A sequence $\{X_1, X_N\}$ is defined to be *monotonic* if ($t(X_i) = CLOSED$ and $h(G, X_i) < h(G, X_{i+1})$) or ($t(X_i) = OPEN$ and $k(G, X_i) < h(G, X_{i+1})$) for all i such that $1 \le i < N$. D* constructs and maintains a monotonic sequence $\{G, X\}$, representing decreasing current or lower-bounded path costs, for each state X that is or was on the *OPEN* list. Given a sequence of states $\{X_1, X_N\}$, state X_i is an *ancestor* of state X_j if $1 \le i < j \le N$ and a *descendant* of X_j if $1 \le j < i \le N$.

For two-state functions involving the goal state, the following shorthand notation is used: $h(X) \equiv h(G, X)$ and $k(X) \equiv k(G, X)$. Likewise, for sequences the notation $\{X\} \equiv \{G, X\}$ is used. The notation $f(\cdot)$ is used to refer to a function independent of its domain.

2.3 Algorithm Description

The Basic D* algorithm consists primarily of two functions: *PROCESS – STATE* and *MODIFY – COST*. *PROCESS – STATE* is used to compute optimal path costs to the goal, and *MODIFY – COST* is used to change the arc cost function $c(\cdot)$ and enter affected states on the *OPEN* list. The algorithms for *PROCESS – STATE* and *MODIFY – COST* are presented below. The embedded routines are $MIN(a, b)$, which returns the minimum of the two scalar values a and b; $LESS(a, b)$, which returns *TRUE* if a is less than b and *FALSE* otherwise; $COST(X)$, which returns $h(X)$ for state X; *MIN – STATE*, which returns the state on the *OPEN* list with minimum $k(\cdot)$ value (*NULL* if

the list is empty); $MIN-VAL$, which returns k_{min} for the $OPEN$ list ($NO-VAL$ if the list is empty); $DELETE(X)$, which deletes state X from the $OPEN$ list and sets $t(X) = CLOSED$; and $INSERT(X, h_{new})$, which computes $k(X) = h_{new}$ if $t(X) = NEW$, $k(X) = MIN(k(X), h_{new})$ if $t(X) = OPEN$, and $k(X) = MIN(h(X), h_{new})$ if $t(X) = CLOSED$, sets $h(X) = h_{new}$ and $t(X) = OPEN$, and places or re-positions state X on the $OPEN$ list sorted by $k(°)$.

The function names $LESS$ and $COST$ are used rather than $<$ and $h(°)$, respectively, since their semantics are redefined in Section 3.3 to operate on vectors rather than scalars.

In function $PROCESS-STATE$ at lines L1 through L3, the state X with the lowest $k(°)$ value is removed from the $OPEN$ list. If X is a $LOWER$ state (i.e., $k(X) = h(X)$), its path cost is optimal since $h(X)$ is equal to the old k_{min}. At lines L8 through L13, each neighbor Y of X is examined to see if its path cost can be lowered. Additionally, neighbor states that are NEW receive an initial path cost value, and cost changes are propagated to each neighbor Y that has a backpointer to X, regardless of whether the new cost is greater than or less than the old. Since these states are descendants of X, any change to the path cost of X affects their path costs as well. The backpointer of Y is redirected, if needed, so that the monotonic sequence $\{Y\}$ is constructed. All neighbors that receive a new path cost are placed on the $OPEN$ list, so that they will propagate the cost changes to their neighbors.

Function: PROCESS-STATE ()

```
L1   X = MIN-STATE( )
L2   if X = NULL then return NO-VAL
L3   k_old = k(X) ; DELETE(X)
L4   if k_old < h(X) then
L5         for each neighbor Y of X:
L6               if t(Y) ≠ NEW and h(Y) ≤ k_old and h(X) > h(Y) + c(Y, X) then
L7                     b(X) = Y; h(X) = h(Y) + c(Y, X)
L8   if k_old = h(X) then
L9         for each neighbor Y of X:
L10              if t(Y) = NEW or
L11                    (b(Y) = X and h(Y) ≠ h(X) + c(X, Y)) or
L12                    (b(Y) ≠ X and h(Y) > h(X) + c(X, Y)) then
L13                          b(Y) = X; INSERT(Y, h(X) + c(X, Y))
L14  else
L15        for each neighbor Y of X:
L16              if t(Y) = NEW or
L17                    (b(Y) = X and h(Y) ≠ h(X) + c(X, Y)) then
L18                          b(Y) = X; INSERT(Y, h(X) + c(X, Y))
L19              else
L20                    if b(Y) ≠ X and h(Y) > h(X) + c(X, Y) and t(X) = CLOSED then
L21                          INSERT(X, h(X))
L22                    else
L23                          if b(Y) ≠ X and h(X) > h(Y) + c(Y, X) and
L24                                t(Y) = CLOSED and h(Y) > k_old then
L25                                      INSERT(Y, h(Y))
L26  return MIN-VAL( )
```

If X is a $RAISE$ state, its path cost may not be optimal. Before X propagates cost changes to its neighbors, its optimal neighbors are examined at lines L4 through L7 to see if $h(X)$ can be reduced. At lines L15 through L18, cost changes

are propagated to *NEW* states and immediate descendants in the same way as for *LOWER* states. If X is able to lower the path cost of a state that is not an immediate descendant (lines L20 and L21), X is placed back on the *OPEN* list for future expansion. This action is required to avoid creating a closed loop in the backpointers [12]. If the path cost of X is able to be reduced by a suboptimal neighbor (lines L23 through L25), the neighbor is placed back on the *OPEN* list. Thus, the update is "postponed" until the neighbor has an optimal path cost. To assist the reader in understanding this cost propagation process, a simplified but less computationally efficient version of *PROCESS – STATE* is included in the Appendix.

In function *MODIFY – COST*, the arc cost function is updated with the changed value. Since the path cost for state Y will change, X is placed on the *OPEN* list. When X is expanded via *PROCESS – STATE*, it computes a new $h(Y) = h(X) + c(X, Y)$ and places Y on the *OPEN* list. Additional state expansions propagate the cost to the descendants of Y.

Function: MODIFY-COST (X, Y, cval)

L1 $c(X, Y) = cval$
L2 if $t(X) = CLOSED$ then $INSERT(X, h(X))$
L3 return $MIN - VAL( )$

The function *MOVE – ROBOT* illustrates how to use *PROCESS – STATE* and *MODIFY – COST* to move the robot from state S through the environment to G along an optimal traverse. At lines L1 through L3 of *MOVE – ROBOT*, $t(°)$ is set to *NEW* for all states, $h(G)$ is set to zero, and G is placed on the *OPEN* list. *PROCESS – STATE* is called repeatedly at lines L5 and L6 until either an initial path is computed to the robot's state (i.e., $t(S) = CLOSED$) or it is determined that no path exists (i.e., $val = NO - VAL$ and $t(S) = NEW$). The robot then proceeds to follow the backpointers in the sequence $\{R\}$ until it either reaches the goal or discovers a discrepancy (lines L10 and L11) between the *sensor measurement* of an arc cost $s(°)$ and the stored arc cost $c(°)$ (e.g., due to a detected obstacle). Note that these discrepancies may occur anywhere, not just in the sequence $\{R\}$. *MODIFY – COST* is called to correct $c(°)$ and place affected states on the *OPEN* list. *PROCESS – STATE* is then called repeatedly at line L13 until $val \geq h(R)$ to propagate costs and compute a possibly new sequence $\{R\}$ to the goal. The robot continues to follow the backpointers in the sequence toward the goal. The function returns *GOAL – REACHED* if the goal is found and *NO – PATH* if it is unreachable.

Function: MOVE-ROBOT (S, G)

L1 for each state X in the graph:
L2 $\quad t(X) = NEW$
L3 $INSERT(G, 0)$
L4 $val = 0$
L5 while $t(S) \neq CLOSED$ and $val \neq NO - VAL$
L6 $\quad val = PROCESS - STATE( )$
L7 if $t(S) = NEW$ then return $NO - PATH$
L8 $R = S$
L9 while $R \neq G$:
L10 $\quad$ for each (X,Y) such that $s(X, Y) \neq c(X, Y)$:
L11 $\quad\quad val = MODIFY - COST(X, Y, s(X, Y))$
L12 $\quad$ while $LESS(val, COST(R))$ and $val \neq NO - VAL$
L13 $\quad\quad val = PROCESS - STATE( )$
L14 $\quad R = b(R)$
L15 return $GOAL - REACHED$

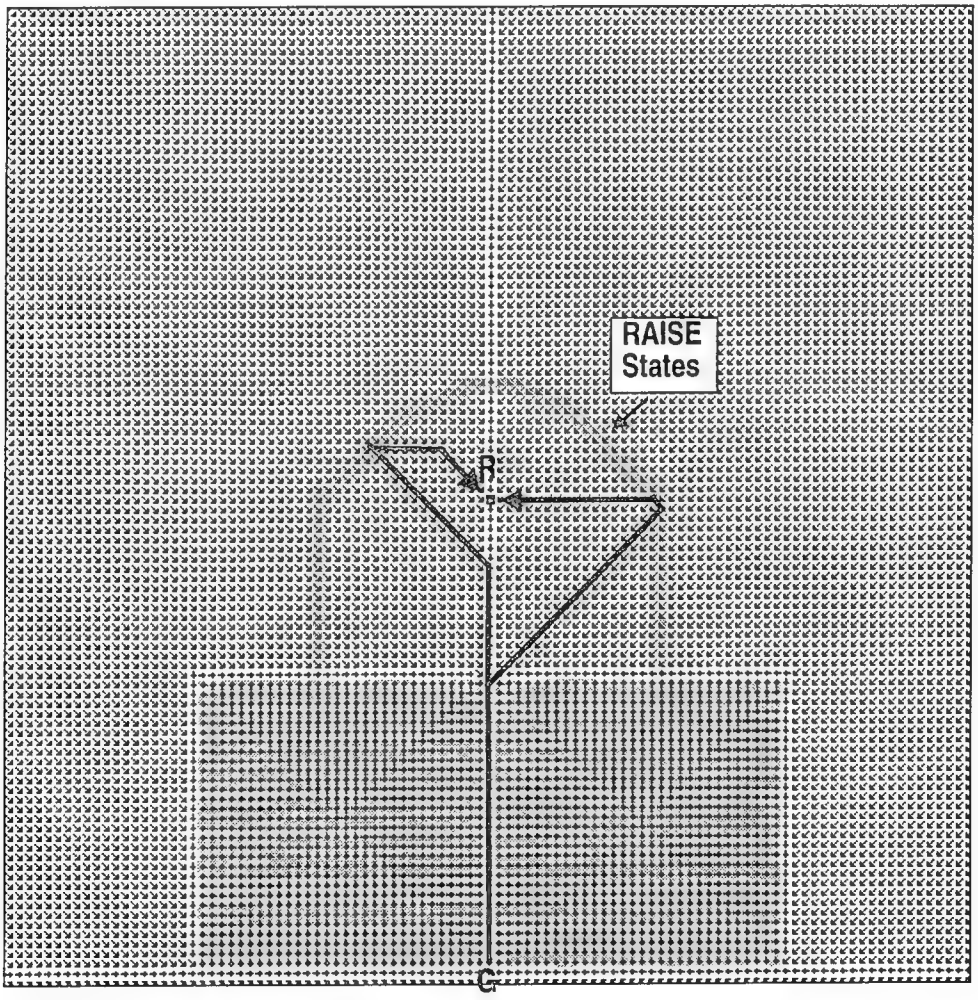
It should be noted that line L7 in *MOVE – ROBOT* only detects the condition that no sequence of arcs exists from the robot's state to the goal, if for example, the graph is disconnected. It does not detect the condition that all paths to the goal are obstructed by obstacles. In order to provide for this capability, obstructed arcs can be assigned a large positive value of *OBSTACLE* and unobstructed arcs can be assigned a small positive value of *EMPTY*. *OBSTACLE* should be chosen such that it exceeds the longest possible sequence of *EMPTY* arcs in the graph. No unobstructed path exists to the goal from S if $h(S) \geq OBSTACLE$ after exiting the loop at line L5. Likewise, no unobstructed path exists to the goal from a state R during the traverse if $h(R) \geq OBSTACLE$ after exiting the loop at line L12.

3.0 The Focussed D* Algorithm

3.1 Intuition

The Basic D* algorithm in the previous section propagates cost changes through the invalidated states without considering which expansions will benefit the robot at its current location. Like A*, D* can use heuristics to focus the search in the direction of the robot and reduce the total number of state expansions. Let the *focussing heuristic* $g(X, R)$ be the estimated path cost from the robot's location R to X. Define a new function, the *estimated robot path cost*, to be $f(X, R) = h(G, X) + g(X, R)$, and sort all *LOWER* states on the *OPEN* list by increasing $f(°)$ value. The function $f(X, R)$ is the estimated path cost from the state R through X to G. Provided that $g(°)$ satisfies the monotone restriction (i.e., $g(G, G) = 0$ and $g(G, X) - g(G, Y) \leq c(Y, X)$ for all (X, Y) such that is $c(Y, X)$ defined), then since $h(G, X)$ is optimal when *LOWER* state X is removed from the *OPEN* list, an optimal path will be computed to R [8].

Figure 4: Focussed *RAISE* State Propagation



In the case of *RAISE* states, the previous $h(°)$ value defines a lower bound on the $h(°)$ values of *LOWER* states they can discover (see Section 2.1); therefore, if the same focussing heuristic $g(°)$ is used for both types of states, the previous $f(°)$ values for the *RAISE* states define lower bounds on the $f(°)$ values for the *LOWER* states they can discover. Thus, if the $f(°)$ values of the *LOWER* states on the *OPEN* list exceed the previous $f(°)$ values of the *RAISE* states, then it is worthwhile processing *RAISE* states to discover (possibly) better *LOWER* states. Based on this reasoning, the *RAISE* states should be sorted on the *OPEN* list by $f(X, R) = k(G, X) + g(X, R)$. But since

$k(G, X) = h(G, X)$ for *LOWER* states, the *RAISE* state definition for $f(°)$ suffices for both kinds of states. To avoid cycles in the backpointers, it should be noted that ties in $f(°)$ are sorted by increasing $k(°)$ on the *OPEN* list [12].

The process can terminate when the lowest value on the *OPEN* list equals or exceeds the robot's path cost, since the subsequent expansions cannot possibly find a *LOWER* state that 1) has a low enough path cost, and 2) is "close" enough to the robot to be able to reduce the robot's path cost when it reaches it through subsequent expansions. Note that this is a more efficient cut-off than the one outlined in Section 2.1, which considers only the first criterion.

Figure 5: Focussed *LOWER* States Reach Robot's State

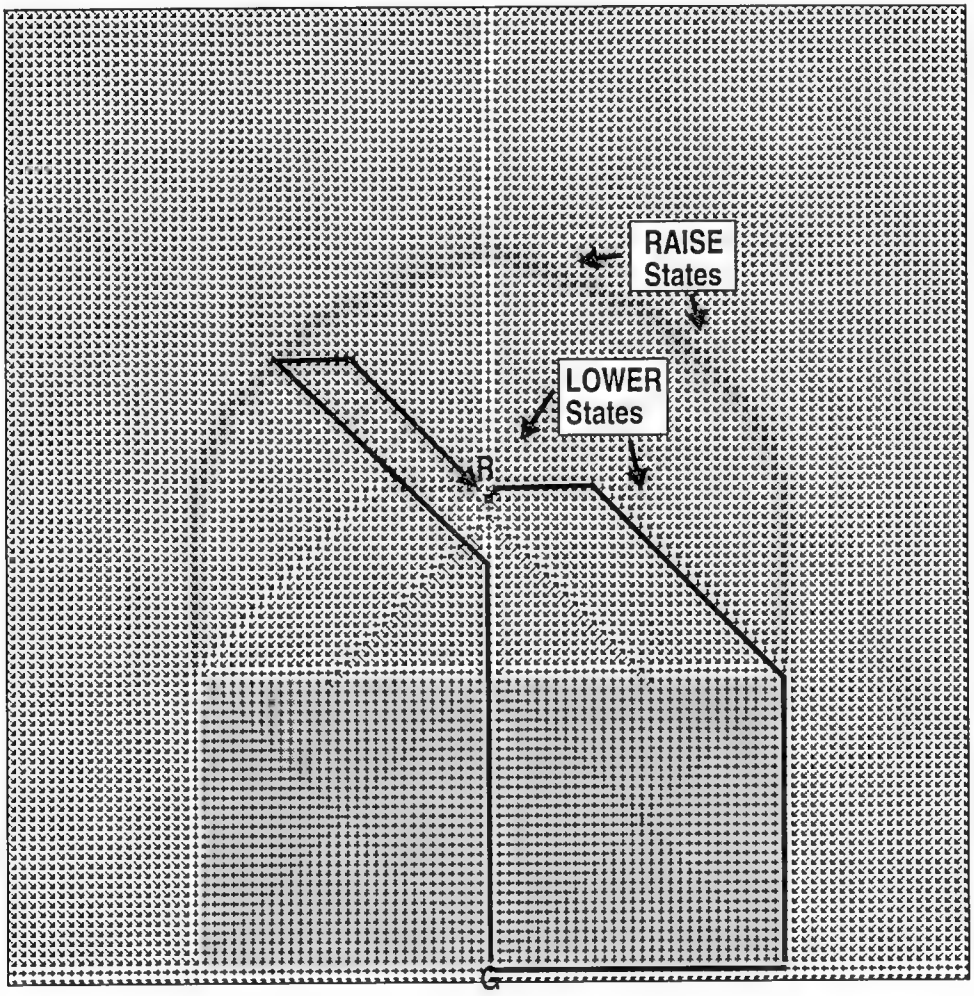


Figure 4 shows the same example as Figure 2, except that a focussed search is used. All states in the *RAISE* state wave front have roughly the same $f(°)$ value. Note that for the two *RAISE* states selected, the arrows from the goal to the *RAISE* states and back to the robot are the same length. The wave front in Figure 4 is more "narrow" than that in Figure 2 since the inclusion of the cost to return to the robot penalizes the wide flanks in Figure 2.

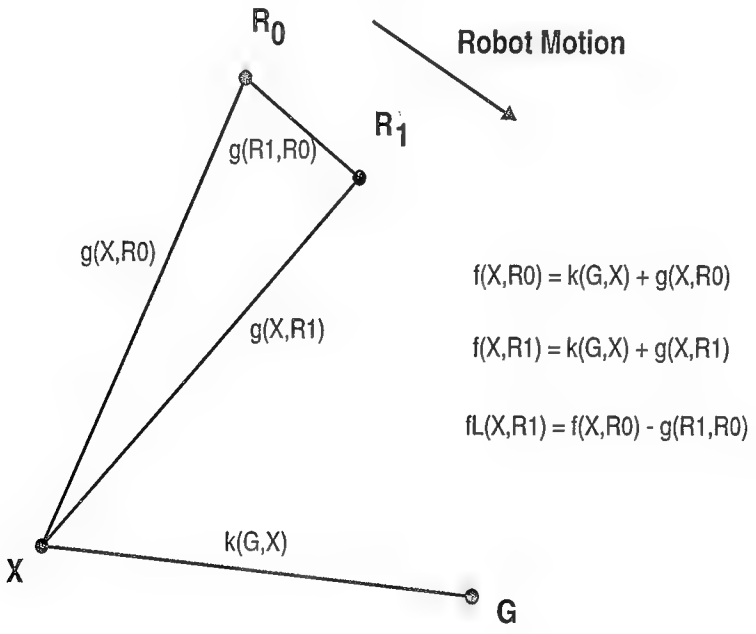
In Figure 5, the *LOWER* states activated by the *RAISE* state wave front have swept in from the outer sides of the obstacles to compute a new, optimal path to the robot. Note that the two wave fronts are narrow and focussed on the robot's location. Again, the lengths of the arrows to the two wave fronts and back to the robot are equal.

Compare Figure 5 to Figure 3. Note that both the *RAISE* and *LOWER* state wave fronts have covered less ground for the focussed search than the unfocussed search in order to compute a new, optimal path to R. Therein is the efficiency of the Focussed D* algorithm.

The problem with focussing the search is that once a new optimal path is computed to the robot's location, it then moves to a new location. If the robot's sensor discovers another arc cost discrepancy, the search should be focussed on the robot's new location. But states already on the *OPEN* list are focussed on the old location and have incorrect $g(°)$ and $f(°)$ values. One solution is to recompute $g(°)$ and $f(°)$ for all states on the *OPEN* list every time the robot moves and new states are to be added via *MODIFY–COST*. This approach is inefficient since it re-sorts the *OPEN* list, requiring at worst $O(N\log N)$ operations, where N is the number of states on the *OPEN* list. Based on empirical evidence, this additional computation more than offsets the savings gained by a focussed search.

The approach in this paper is to take advantage of the fact that the robot generally moves only a few states between calls to *MODIFY–COST*, so the $g(°)$ and $f(°)$ values have only a small amount of error. Assume that state X is placed on the *OPEN* list when the robot is at location R_0 (see Figure 6). Its $f(°)$ value at that point is $f(X, R_0)$. If the robot moves to location R_1, we could calculate $f(X, R_1)$ and adjust its position on the *OPEN* list. To avoid this computational cost, we compute a lower bound on $f(X, R_1)$ given by $f_L(X, R_1) = f(X, R_0) - g(R_1, R_0)$. $f_L(X, R_1)$ is a lower bound on $f(X, R_1)$ since it assumes the robot moved in the "direction" of state X, thus subtracting the motion from $g(X, R_0)$. If X is adjusted on the *OPEN* list by $f_L(X, R_1)$, then since $f_L(X, R_1)$ is a lower bound on $f(X, R_1)$, X will be selected for expansion before or when it is needed. At the time of expansion, the true $f(X, R_1)$ value is computed, and X is placed back on the *OPEN* list by $f(X, R_1)$.

Figure 6: Computing a Lower Bound on $f(°)$ for Robot Motion



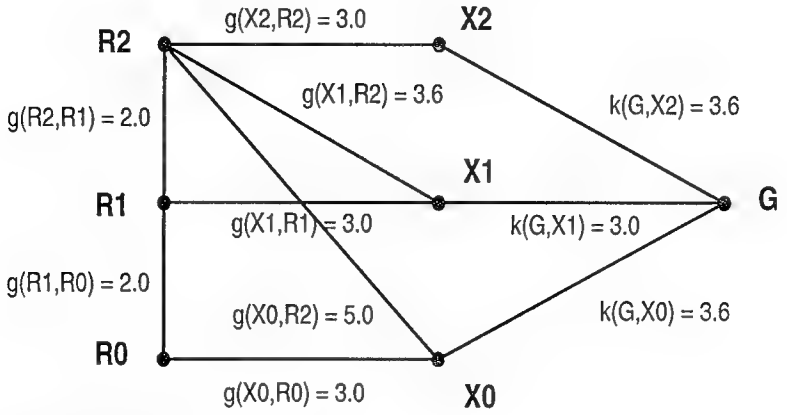
At first this appears worse, since the *OPEN* list is first re-sorted by $f_L(°)$ and then partially adjusted to replace the $f_L(°)$ values with the correct $f(°)$ values. But since $g(R_1, R_0)$ is subtracted from *all* states on the *OPEN* list, the ordering is preserved, and the list need not be re-sorted. Furthermore, the first step can be avoided altogether by *adding* $g(R_1, R_0)$ to the states to be inserted on the *OPEN* list rather than *subtracting* it from those already on the list, thus preserving the relative ordering between states already on the list and states about to be added. Therefore, the only remaining computation is the adjustment step. But this step is needed only for those states that show promise for reaching the robot's location. For typical problems, this amounts to fewer than 2% of the states on the *OPEN* list (see Section 5.0).

3.2 Definitions

To formalize this notion, states are sorted on the *OPEN* list by a *biased* $f(°)$ value, given by $f_B(X, R_i)$, where X is the state on the *OPEN* list and R_i is the robot's state at the time X was inserted on the *OPEN* list. Let $\{R_0, R_1, \ldots, R_N\}$ be

the sequence of states occupied by the robot when states are added to the *OPEN* list via *MODIFY – COST*. The value of $f_B(°)$ is given by $f_B(X, R_i) = f(X, R_i) + d(R_i, R_0)$, where $d(°)$ is the *accrued bias* function given by $d(R_i, R_0) = g(R_1, R_0) + g(R_2, R_1) + \ldots + g(R_i, R_{i-1})$ if $i > 0$ and $d(R_0, R_0) = 0$ if $i = 0$. The *OPEN* list states are sorted by increasing $f_B(°)$ value, with ties in $f_B(°)$ ordered by increasing $f(°)$, and ties in $f(°)$ ordered by increasing $k(°)$. Ties in $k(°)$ are ordered arbitrarily. Thus, a vector of values $\langle f_B(°), f(°), k(°) \rangle$ is stored with each state on the list.

Figure 7: Computing Bias Values for $f(°)$



Proper ordering:

f(X1,R2) = k(G,X1) + g(X1,R2) = 6.6

f(X2,R2) = k(G,X2) + g(X2,R2) = 6.6

f(X0,R2) = k(G,X0) + g(X0,R2) = 8.6

Initial bias ordering:

fB(X0,R0) = k(G,X0) + g(X0,R0) = 6.6

fB(X1,R1) = k(G,X1) + g(X1,R1) + g(R1,R0) = 8.0

fB(X2,R2) = k(G,X2) + g(X2,R2) + g(R2,R1) + g(R1,R0) = 10.6

Adjusted bias ordering:

fB(X1,R2) = k(G,X1) + g(X1,R2) + g(R2,R1) + g(R1,R0) = 10.6

fB(X2,R2) = k(G,X2) + g(X2,R2) + g(R2,R1) + g(R1,R0) = 10.6

fB(X0,R2) = k(G,X0) + g(X0,R2) + g(R2,R1) + g(R1,R0) = 12.6

Consider the example shown in Figure 7. The robot starts at location R_0 and places state X_0 on the *OPEN* list. It then moves to R_1 and places X_1 on the list. Finally, it moves to R_2 and inserts X_2. At location R_2, it expands the three states on the *OPEN* list. If the $f(°)$ values of the three states were recomputed with the robot at location R_2, the proper ordering on the *OPEN* list would be $\{X_1, X_2, X_0\}$. Note that the tie in $f(°)$ values for X_1 and X_2 is broken in favor of lower $k(°)$. Since the states were placed on the *OPEN* list at different locations, the ordering by $f_B(°)$ value is $\{X_0, X_1, X_2\}$. The first state removed from the list is X_0. It's $f_B(°)$ value is changed from $f_B(X_0, R_0)$ to $f_B(X_0, R_2)$ (i.e., 6.6 to 12.6), and it is put back on the list according to the adjusted value. The next state removed is X_1. It's $f_B(°)$ value is changed from $f_B(X_1, R_1)$ to $f_B(X_1, R_2)$ (i.e., 8.0 to 10.6) and is placed back on the list. The next state removed is X_1 once again, since it has the same $f_B(°)$ and $f(°)$ values as X_2 but has a lower $k(°)$ value. Since it has the proper $f_B(°)$ value (i.e., computed at the robot's current location, R_2), the state is expanded. Note that this is the first state that should be expanded. The next state removed is X_2. Since it already has the proper $f_B(°)$ value, it is expanded. Finally, X_0 is removed. Since its $f_B(°)$ value was properly adjusted for the new robot location, it is expanded. Thus, the three states are expanded in the proper order.

For a graph representing an eight-connected Cartesian array of locations, a good focussing heuristic that satisfies the monotone restriction is the minimum arc distance heuristic. If the Cartesian array is indexed by (i, j), let (x_i, x_j) be the (i, j) coordinates of state X. Let C_{min} be the minimum $c(°)$ cost in the array, after all arc costs are normalized to the length of a non-diagonal arc. Let d_i be $|x_i - y_i|$ and d_j be $|x_j - y_j|$. Then $g(X, Y) = C_{min}\left(\sqrt{2}d_j + d_i - d_j\right)$ if $d_i \geq d_j$ and $g(X, Y) = C_{min}\left(\sqrt{2}d_i + d_j - d_i\right)$ if $d_i < d_j$.

Let R_{curr} be the most recent robot state at which discrepancies were discovered between the sensor data and map, and let R_{prev} be the previous such state. Both are initialized to the robot's start state. Define the *robot state* function $r(X)$, which returns the robot's state when X was last inserted or adjusted on the *OPEN* list. The parameter d_{curr} is the accrued bias from the robot's start state to its current state; it is shorthand for $d(R_{curr}, R_0)$ and is initialized to $d_{curr} = d(R_0, R_0) = 0$. Let f_{min} be the minimum $f(°)$ value on the *OPEN* list and k_{val} be its corresponding $k(°)$ value. The following shorthand notation is used for $f_B(°)$, $f(°)$, and $g(°)$: $f_B(X) \equiv f_B(X, r(X))$, $f(X) \equiv f(X, r(X))$, and $g(X) \equiv g(X, r(X))$.

3.3 Algorithm Extension

Most of the extensions are confined to the functions for cost comparisons and management of the *OPEN* list; therefore, the functions *COST*, *LESS*, *INSERT*, *MIN – STATE*, and *MIN – VAL* are affected. Instead of returning $h(R)$ for a robot state R, *COST*(R) returns the vector of values $\langle f(R, R_{curr}), h(R)\rangle$. Instead of comparing two scalars, the function *LESS*(a, b) takes a vector of values $\langle a_1, a_2\rangle$ for a and a vector $\langle b_1, b_2\rangle$ for b. *LESS* returns *TRUE* if $a_1 < b_1$ or $(a_1 = b_1$ and $a_2 < b_2)$; otherwise, it returns *FALSE*.

Before redefining *INSERT*, *MIN – STATE*, and *MIN – VAL*, two new embedded functions are introduced. *PUT – STATE*(X) sets $t(X) = OPEN$ and inserts X on the *OPEN* list according to the vector $\langle f_B(X), f(X), k(X)\rangle$, and *GET – STATE* returns the state on the *OPEN* list with minimum vector value (*NULL* if the list is empty).

The redefined *INSERT* function is given below. At line L1, the robot's state R, which is manipulated in *MOVE – ROBOT*, is saved as the new focal point for the search. At lines L2 through L4, the current state of the robot is examined to see if the robot has moved since the last time insertions were made to the *OPEN* list. If so, the bias term, $d(°)$, is updated by adding a lower bound on the cost from the robot's previous state to the current $(d(R_{curr}, R_0) = d(R_{prev}, R_0) + g(R_{curr}, R_{prev}))$. The values for $h(X)$ and $k(X)$ are determined at lines L5 through L11. The remaining two values in the vector are computed at line L12, and the state is inserted at line L13.

Function: INSERT (X, hnew)

L1 $R_{curr} = R$
L2 if $R_{curr} \neq R_{prev}$ then
L3 $\quad d_{curr} = d_{curr} + g(R_{curr}, R_{prev})$
L4 $\quad R_{prev} = R_{curr}$
L5 if $t(X) = NEW$ then $k(X) = h_{new}$
L6 else
L7 $\quad$ if $t(X) = OPEN$ then
L8 $\quad\quad k(X) = MIN(k(X), h_{new})$
L9 $\quad\quad DELETE(X)$
L10 $\quad$ else $k(X) = MIN(h(X), h_{new})$
L11 $h(X) = h_{new}$; $r(X) = R_{curr}$
L12 $f(X) = k(X) + g(X)$; $f_B(X) = f(X) + d_{curr}$
L13 $PUT - STATE(X)$

The function $MIN-STATE$, given below, returns the state on the $OPEN$ list with minimum $f(°)$ value. In order to do this, the function retrieves the state on the $OPEN$ list with lowest $f_B(°)$ value. If the state was placed on the $OPEN$ list when the robot was at a previous location (line L2), then it is re-inserted on the $OPEN$ list at line L3. This operation has the effect of correcting the state's bias term using the robot's current state while leaving the state's $h(°)$ and $k(°)$ values unchanged. $MIN-STATE$ continues to retrieve states from the $OPEN$ list until it finds one that has an updated bias term (i.e., it was placed on the $OPEN$ list with the robot at its current state).

Function: MIN-STATE ()

L1 while $X = GET-STATE(\ ) \neq NULL$

L2 if $r(X) \neq R_{curr}$ then

L3 $h_{new} = h(X)$; $h(X) = k(X)$; $DELETE(X)$; $INSERT(X, h_{new})$

L4 else return X

L5 return $NULL$

The redefined $MIN-VAL$ function, given below, returns the $f(°)$ and $k(°)$ values of the state on the $OPEN$ list with minimum $f(°)$ value.

Function: MIN-VAL ()

L1 $X = MIN-STATE(\ )$

L2 if $X = NULL$ then return $NO-VAL$

L3 else return $\langle f(X), k(X) \rangle$

The functions $PROCESS-STATE$ and $MODIFY-COST$ are syntactically unchanged from their descriptions given in Section 2.3. But since both functions conclude by returning the result of a call to $MIN-VAL$, they also return the vector of values $\langle f_{min}, k_{val} \rangle$ instead of just k_{min}. This vector is assigned to val in the routine $MOVE-ROBOT$ and is used by the redefined $LESS$ function to determine if $PROCESS-STATE$ has been called enough times to guarantee optimality. Since $R = R_{curr}$ for a robot state R undergoing path recalculations, then $g(R, R) = 0$ and $f(R, R) = h(R)$. Therefore, optimality is guaranteed for a state R, if $f_{min} > h(R)$ or ($f_{min} = h(R)$ and $k_{val} \geq h(R)$).

4.0 Examples

This section presents two examples that illustrate the operation of the D* algorithm. The first compares the Basic D* algorithm to Focussed D*, and the second compares planning with complete information to optimistic information.

4.1 D* Comparisons

Figure 8 shows a cluttered 100 x 100 state environment. The robot starts at state S and moves to state G. All of the obstacles, shown in black, are unknown before the robot starts its traverse, and the map contains only $EMPTY$ arcs. The robot is point-sized and is equipped with a 10-state radial field-of-view sensor.

Figure 8: Cluttered Environment

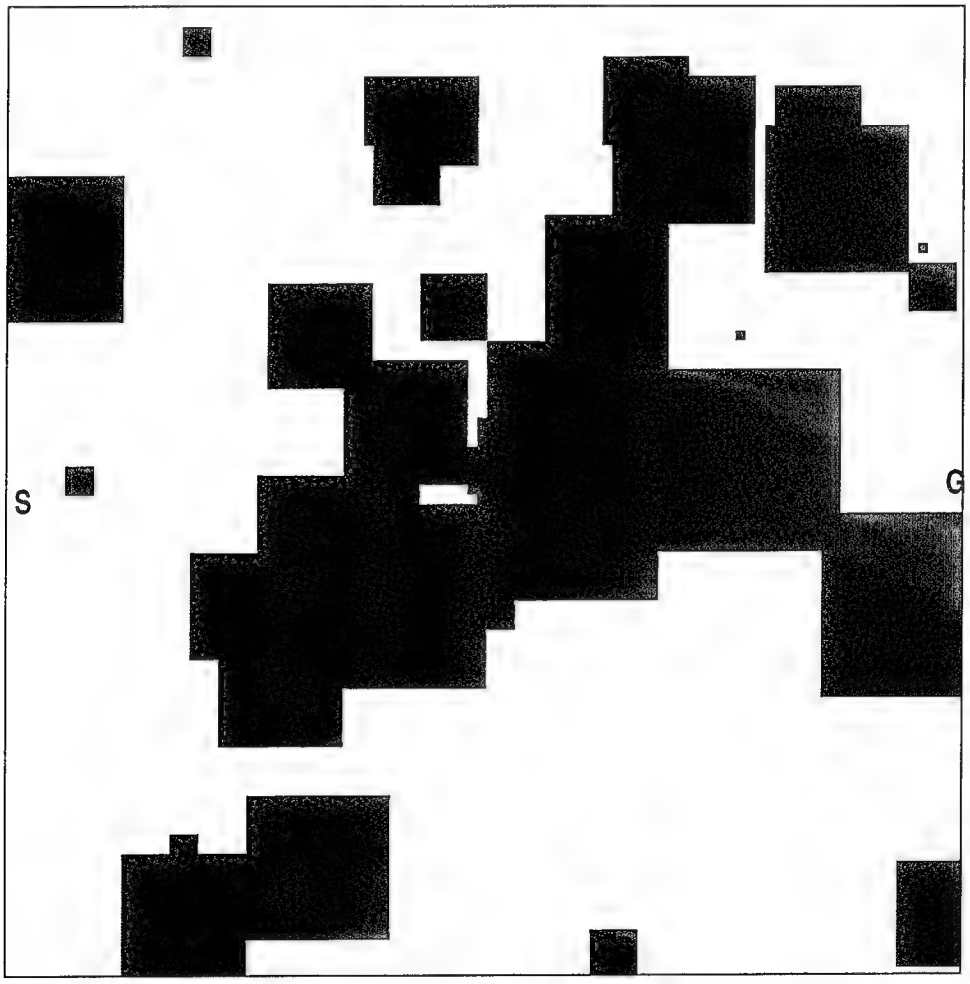


Figure 9: Basic D* Algorithm

Figure 9 shows the robot's traverse from S to G using the Basic D* algorithm. The traverse is shown as a black curve with white arrows. As the robot moves, its sensor detects the unknown obstacles. Detected obstacles are shown in grey with black arrows. Obstacles that remain unknown after the traverse are shown in solid black or black with white arrows. The arrows show the final cost field for all states examined during the traverse. Note that most of the states are examined at least once by the algorithm.

Figure 10: Focussed D* Algorithm

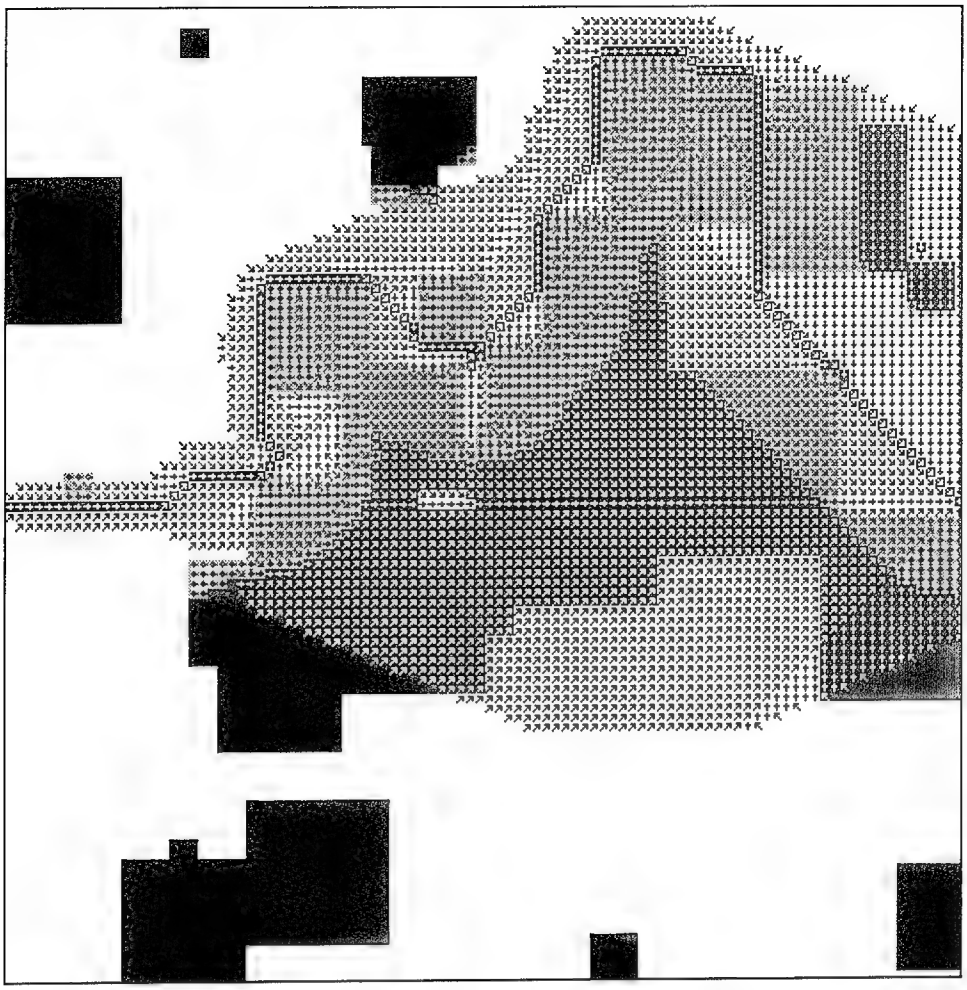
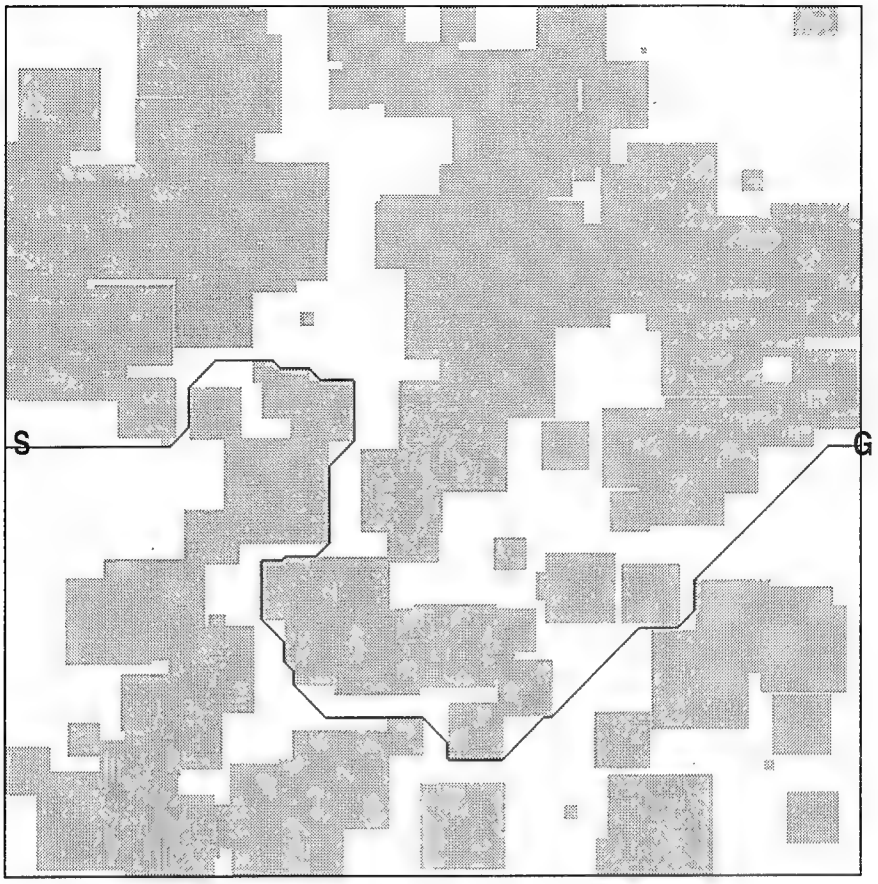


Figure 10 shows the robot's traverse using the Focussed D* algorithm. The number of *NEW* states examined is fewer than Basic D*, since the Focussed D* algorithm focuses the initial path calculation and subsequent cost updates on the robot's location. Note that even for those states examined by the algorithm, fewer of them end up with optimal paths to the goal. Finally, note that the two trajectories are not fully equivalent. This occurs because the lowest-cost traverse is not unique, and the two algorithms break ties in the path costs arbitrarily.

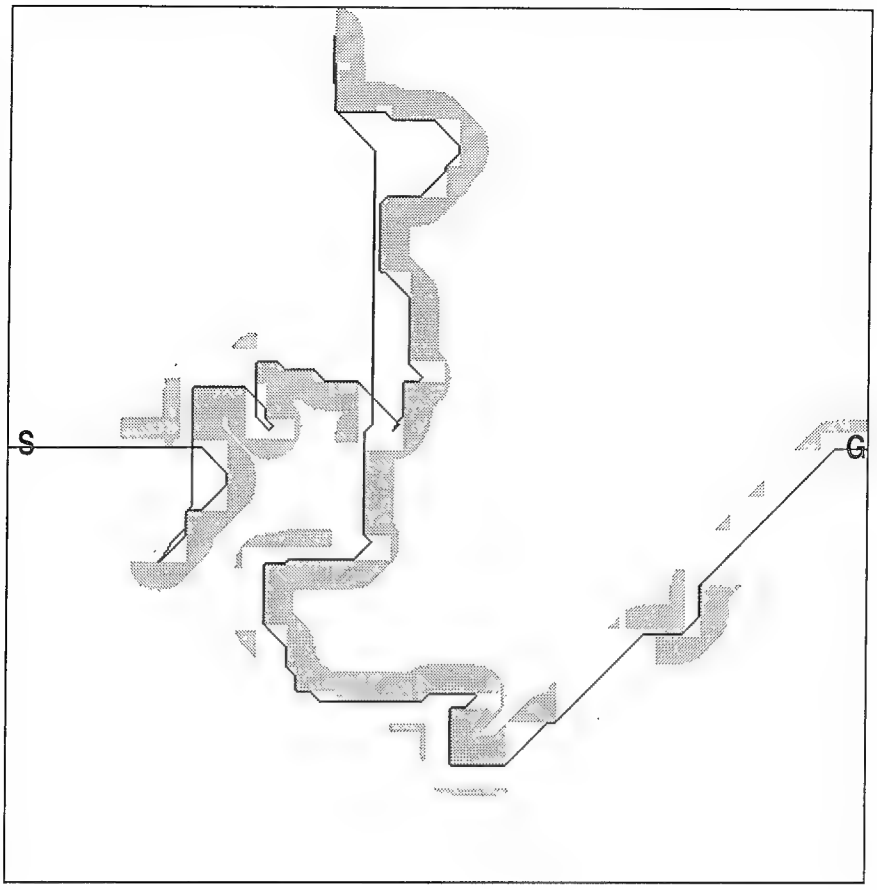
Figure 11: Omniscient Optimal Traverse



4.2 A Priori Map Information

Figure 11 shows a 450 x 450 state environment cluttered with grey obstacles. The backpointers were omitted for clarity. The black curve shows the optimal traverse to the goal, given the case where the robot knows about the obstacles before it begins its traverse. This traverse is known as *omniscient optimal*, since it is equivalent to the optimal path to the goal, given complete and accurate arc cost information. Figure 12 shows planning in the same environment where none of the obstacles are stored in the map a priori. The portions of the obstacles detected by the robot's 15-state radial sensor are shown. This traverse is known as *optimistic optimal*, since the robot assumes no obstacles exist unless they are detected by its sensor. This traverse is nearly two times longer than omniscient optimal; however, it is still optimal given the initial map and the sensor information as it is acquired.

Figure 12: Optimistic Optimal Traverse



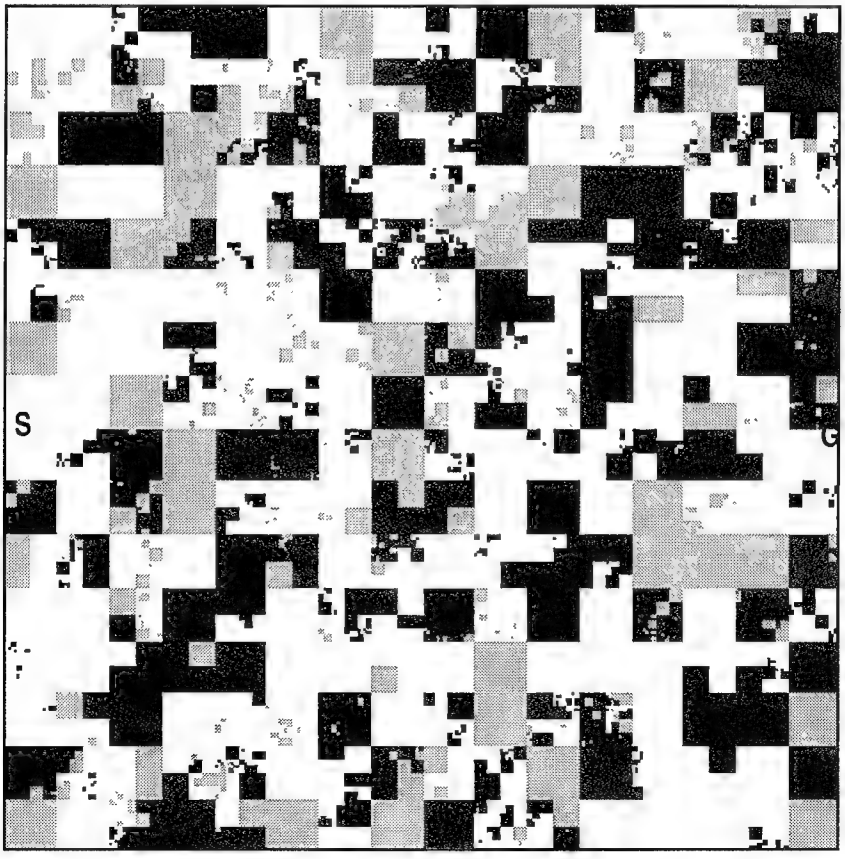
5.0 Experimental Results

Four algorithms were tested to verify optimality and to compare run-time and memory usage results. The first algorithm, the Brute-Force Replanner (BFR), initially plans a single path from the goal to the start state. The robot proceeds to follow the path until its sensor detects an error in the map. The robot updates the map, plans a new path from the goal to its current location using a focussed A* search [8], and repeats until the goal is reached. The focussing heuristic used is the minimum arc distance described in Section 3.2.

The second and third algorithms, Basic D* (BD*) and Focussed D* with Minimal Initialization (FD*M), are described in Sections 2.0 and 3.0, respectively. The fourth algorithm, Focussed D* with Full Initialization (FD*F), is the same as FD*M except that the path costs are propagated to all states in the planning space, which is assumed to be finite, during the initial path calculation, rather than terminating when the path reaches the robot's start state.

The four algorithms were compared on planning problems of varying size. Each environment was square, consisting of a start state in the center of the left wall and a goal state in center of the right wall. Each environment consisted of a mix of map obstacles known to the robot before the traverse and unknown obstacles measurable by the robot's sensor. The sensor used was omnidirectional with a 10-state radial field of view. Figure 13 shows an environment model with approximately 100,000 states. The known obstacles are shown in grey and the unknown obstacles in black.

Figure 13: Typical Environment for Comparison



The results for environments of 10,000, 100,000, and 1,000,000 states are shown in Tables 1, 2, and 3, respectively. The reported times are CPU time for a Sun Microsystems SPARC-10 processor. For each environment size, the four algorithms were compared on five randomly-generated environments, and the results were averaged. The algorithms were coupled so that ties in paths costs would be broken in the same way, and the traverses would be identical. The

off-line time is the CPU time required to compute the initial path from the goal to the robot, or in the case of FD*F, from the goal to all states in the environment. This operation is "off-line" since it could be performed in advance of robot motion if the initial map were available. The *on-line time* is the total CPU time for all replanning operations needed to move the robot from the start to the goal. It constitutes the CPU time needed to keep the robot moving toward the goal, and hence is "on-line". The *memory %* is the percentage of map states examined across all planning and replanning operations. In the case of BFR, it is the largest percentage of map states examined during any single planning or replanning operation, since path costs are not retained from one replanning operation to the next. This parameter corresponds to memory usage if states are dynamically allocated. The *on-line %* is the percentage of *OPEN* list states that are re-sorted due to robot motion averaged across all replanning operations and is applicable only to the Focussed D* algorithm in both forms. It is a metric for how good the lower bound approximation $f_L(°)$ is to $f(°)$.

The results for each algorithm are highly dependent on the complexity of the environment, including the number, size, and placement of the obstacles, and the ratio of known to unknown obstacles. For the test cases examined, all variations of D* outperformed BFR in on-line time, reaching a speedup factor of nearly 300 for large environments. Generally, the performance gap increased as the size of the environment increased. The FD*M algorithm resulted in lower off-line times and higher on-line times than BD*. Focussing the search enables a rapid start due to fewer state expansions, but many of the unexplored states must be examined anyway during the replanning process resulting in a longer execution time. Thus, FD*M is the best algorithm if the user wants the robot to begin moving as soon as possible and can afford to pay with more processing during this motion. If the user wants to minimize on-line time at the expense of off-line time, then FD*F is the best algorithm. In this algorithm, path costs to all states are computed initially and only the cost propagations are focussed. Note that FD*F resulted in lower on-line times and higher off-line times than BD*.

Thus, the Focussed D* algorithm can be configured to outperform Basic D* in either the off-line or on-line portion of the operation, depending on the requirements of the task. Note also that the total runtime, given by off-line plus on-line time, is less for FD*M than BD*, even for environments as small as 10,000 states. The disparity widens for larger environments. As a general strategy, focussing the search is a good idea; the only issue is how the computational load should be distributed.

As an aside, the off-line times for all three variations of D* can be reduced by using A*, either focussed or unfocussed, whichever matches the D* algorithm, during the initial planning process, since there are no cost updates to propagate. The A* algorithm has less overhead than D* and is faster per state expansion.

FD*M offers the additional advantage of lower memory usage than BD* if memory is allocated only as needed. Its memory usage is roughly equivalent to that of the BFR, which de-allocates visited states after each replanning operation. FD*F allocates all possible states and is therefore memory intensive.

Table 1: Results for 10,000-State Environments

	Focussed D* with Full Init	Focussed D* with Min Init	Basic D*	Brute-Force Replanner
Off-line Time	1.70 sec	0.15 sec	1.02 sec	0.09 sec
On-line Time	0.90 sec	1.43 sec	1.31 sec	13.07 sec
Memory %	100%	38.7%	87.7%	31.6%
On-line %	1.66%	1.69%	N/A	N/A

Table 2: Results for 100,000-State Environments

	Focussed D* with Full Init	Focussed D* with Min Init	Basic D*	Brute-Force Replanner
Off-line Time	20.58 sec	0.70 sec	12.55 sec	0.41 sec
On-line Time	9.22 sec	17.62 sec	16.94 sec	11.86 min
Memory %	100%	50.4%	83.6%	43.5%
On-line %	0.55%	0.85%	N/A	N/A

Table 3: Results for 1,000,000-State Environment

	Focussed D* with Full Init	Focussed D* with Min Init	Basic D*	Brute-Force Replanner
Off-line Time	221.72 sec	8.68 sec	129.08 sec	4.82 sec
On-line Time	10.26 sec	37.34 sec	21.47 sec	50.63 min
Memory %	100%	15.6%	76.2%	25.3%
On-line %	0.54%	1.29%	N/A	N/A

6.0 Conclusions

This paper presents the D* algorithm for real-time path replanning. The algorithm computes an initial path from the goal state to the start state and then efficiently modifies this path during the traverse as arc costs change. The algorithm is guaranteed to produce an optimal traverse, meaning that an optimal path to the goal is followed at every state in the traverse, assuming all known information at each step is correct. D* is far more efficient than the brute-force path planner. The focussed version of D* outperforms the basic version, and it offers the user the option of distributing the computational load amongst the on- and off-line portions of the operation, depending on the task requirements. Furthermore, the focussed version can be configured to be less memory intensive and is therefore better for large environments.

D* is a very general algorithm and can be applied to problems in artificial intelligence other than robot motion planning. In its most general form, D* can handle any path cost optimization problem where the cost parameters change during the traverse of the solution. D* is most efficient when these changes are detected near the current starting point in the search space, which is the case with a robot equipped with an on-board sensor.

Acknowledgments

The author thanks Barry Brumitt and Jay Gowdy for feedback on the use of the algorithm and the entire Unmanned Ground Vehicle (UGV) project at CMU for providing a vehicle testbed.

References

[1] Boult, T., "Updating Distance Maps when Objects Move", Proc. of the SPIE Conference on Mobile Robots, 1987.

[2] Goto, Y., Stentz, A., "Mobile Robot Navigation: The CMU System", IEEE Expert, Vol. 2, No. 4, Winter, 1987.

[3] Jarvis, R. A., "Collision-Free Trajectory Planning Using the Distance Transforms", Mechanical Engineering Trans. of the Institution of Engineers, Australia, Vol. ME10, No. 3, September, 1985.

[4] Korf, R. E., "Real-Time Heuristic Search: First Results", Proc. Sixth National Conference on Artificial Intelligence, July, 1987.

[5] Latombe, J.-C., "Robot Motion Planning", Kluwer Academic Publishers, 1991.

[6] Lozano-Perez, T., "Spatial Planning: A Configuration Space Approach", IEEE Transactions on Computers, Vol. C-32, No. 2, February, 1983.

[7] Lumelsky, V. J., Stepanov, A. A., "Dynamic Path Planning for a Mobile Automaton with Limited Information on the Environment", IEEE Transactions on Automatic Control, Vol. AC-31, No. 11, November, 1986.

[8] Nilsson, N. J., "Principles of Artificial Intelligence", Tioga Publishing Company, 1980.

[9] Pirzadeh, A., Snyder, W., "A Unified Solution to Coverage and Search in Explored and Unexplored Terrains Using Indirect Control", Proc. of the IEEE International Conference on Robotics and Automation, May, 1990.

[10] Ramalingam, G., Reps, T., "An Incremental Algorithm for a Generalization of the Shortest-Path Problem", University of Wisconsin Technical Report #1087, May, 1992.

[11] Samet, H., "An Overview of Quadtrees, Octrees and Related Hierarchical Data Structures", in NATO ASI Series, Vol. F40, Theoretical Foundations of Computer Graphics, Berlin: Springer-Verlag, 1988.

[12] Stentz, A., "Optimal and Efficient Path Planning for Unknown and Dynamic Environments", Carnegie Mellon Robotics Institute Technical Report CMU-RI-TR-93-20, August, 1993.

[13] Stentz, A., "Optimal and Efficient Path Planning for Partially-Known Environments", Proc. of the IEEE International Conference on Robotics and Automation, May, 1994.

[14] Trovato, K. I., "Differential A*: An Adaptive Search Method Illustrated with Robot Path Planning for Moving Obstacles and Goals, and an Uncertain Environment", Journal of Pattern Recognition and Artificial Intelligence, Vol. 4, No. 2, 1990.

[15] Zelinsky, A., "A Mobile Robot Exploration Algorithm", IEEE Transactions on Robotics and Automation, Vol. 8, No. 6, December, 1992.

Appendix

The function $PROCESS-STATE'$, given below, is more elegant and lucid than $PROCESS-STATE$ given in Section 2.3. After X is removed from the $OPEN$ list at lines L1 through L3, a single pass is made over its neighbors. The state X, regardless of its type, spreads its cost update to its NEW neighbors and immediate descendants at lines L5 and L6. At lines L8 through L12, X is able to reduce the path cost of neighbor Y. If X is a $RAISE$ state (i.e., $h(X) > k_{old}$), then its path cost may not be optimal; therefore, X is placed back on the $OPEN$ list to "postpone" the reduction until it has an optimal path cost. If X is a $LOWER$ state (i.e., $h(X) = k_{old}$), its path cost is optimal; therefore, the path cost of Y is reduced, its backpointer is redirected, and it is placed on the $OPEN$ list to propagate the change. Lines L14 through L18 handle the symmetric case where Y is able to reduce the path cost of neighbor X. Again, the parameter k_{old} is the dividing line between optimal and suboptimal path costs.

Thus, both $RAISE$ and $LOWER$ states spread path cost changes to their descendants and NEW states, but only optimal states are permitted to reduce the path costs of other states and redirect their backpointers. Optimal states are either $LOWER$ states or $CLOSED$ states with path cost values less than or equal to k_{old}. Suboptimal states are put back on the $OPEN$ list until they become optimal.

Function: PROCESS-STATE' ()

```
L1   X = MIN-STATE( )
L2   if X = NULL then return NO-VAL
L3   k_old = k(X) ; DELETE(X)
L4   for each neighbor Y of X:
L5        if t(Y) = NEW or (b(Y) = X and h(Y) ≠ h(X) + c(X, Y)) then
L6             b(Y) = X; INSERT(Y, h(X) + c(X, Y))
L7        else
L8             if b(Y) ≠ X and h(Y) > h(X) + c(X, Y) then
L9                  if h(X) > k_old then
L10                      if t(X) = CLOSED then INSERT(X, h(X))
L11                 else
L12                      b(Y) = X; INSERT(Y, h(X) + c(X, Y))
L13            else
L14                 if b(Y) ≠ X and h(X) > h(Y) + c(Y, X) then
L15                      if h(Y) > k_old then
L16                           if t(Y) = CLOSED then INSERT(Y, h(Y))
L17                      else
L18                           b(X) = Y; INSERT(X, h(Y) + c(Y, X))
L19  return MIN-VAL( )
```

Although $PROCESS-STATE'$ examines the neighbors of X only once rather than twice, it is less computationally efficient than $PROCESS-STATE$, because the expanded state X can spread its path cost change before it is reduced by optimal neighbors. This condition is detected, and X is placed back on the $OPEN$ list to propagate the correct values. These additional $OPEN$ list insertions are expensive compared to the cost of examining a state's neighbors more than once.